Smart Education System: A Full-Stack E-Learning Platform with Embedded Generative AI, Multimodal Content, And Learning Analytics

Augustine Fletcher A. S[1], Sharath S[2], Saikarthik M[3]
[1,2,3]*Electronics and Communication Engineering, SRM Institute of Science and Technology, Vadapalani Campus, Chennai, India*

***Abstract*—We started this project after noticing something annoying — every e-learning platform we used kept the study material and the AI help completely separate. You read a PDF, you get stuck, you open ChatGPT in another tab, paste the content, get an answer, come back. It works, but it is clunky and the instructor has zero visibility into any of it.**

So, we built one platform that keeps everything together. Students can browse subject materials, watch videos, and use AI tools like summarization, quiz generation, and diagram creation without ever leaving the page. Teachers get an upload workflow. Everyone gets a dashboard with XP and activity tracking.

We used Next.js for the full stack, MongoDB for data, and Cloudinary for media storage. For AI we picked Google Gemini — mainly because it has a free API, which mattered for a student project with no budget. All API calls happen server-side so keys never reach the browser. Building it was not always smooth. Gemini's JSON mode occasionally returned output wrapped in markdown blocks instead of raw JSON, which broke our quiz renderer until we added a sanitization step. Cloudinary uploads failed silently when we ran them in parallel with database writes. JWT tokens were inconsistent across routes for a while and caused authenticated users to get 401 errors on the dashboard — that one took us almost two days to track down.

This paper covers the architecture, how we handled these problems, what the system looks like now, and what we would do differently.

***Index Terms*—Smart education, e-learning, generative AI, Next.js, learning analytics, multimodal content, full-stack web application.**

I. INTRODUCTION

A. Motivation and Problem Statement

When we were looking at existing e-learning platforms at the start of this project, the pattern was almost always the same. Content delivery on one side, AI assistance somewhere else entirely. Students jump between tabs constantly. There is no record of what help they sought, no connection to the actual course material, and instructors cannot see any of it [9].

That bothered us. Not in an abstract way — we experienced it ourselves preparing for exams. You have a PDF open; you do not understand a concept, you copy text into a chatbot, get an explanation, come back. It works but it feels broken. The assistance has no context about the course. The instructor has no idea whether students are using AI to actually learn or just to copy answers.

LLMs are good enough now that summarization, quiz generation, and diagram synthesis are genuinely useful through API calls [2], [3], [6]. The harder problem is not whether AI can help — it clearly can. The harder problem is building the plumbing that embeds it properly inside an authenticated system where interactions are logged, API keys are protected, and the output is structured enough to actually render in a UI without falling apart.

That is what this project is about.

B. Objectives and Scope

We had four goals going in. First, deliver subject materials and videos organized by learning track. Second, give teachers a proper upload and publishing workflow backed by cloud storage. Third, run all generative AI features server-side so API credentials never touch the browser. Fourth, track engagement through activity logs, XP points, and leaderboards.

We are three final-year students. The scope reflects that. We built what we could build well in the time we had, tested it properly, and we are honest in this paper

about what we did not finish and what we would do differently with more time.

C. Contributions

- A working full-stack implementation using Next.js, MongoDB, and Cloudnary with role-based access for students, teachers, and admins.
- A server-side Gemini integration with fallback models, response caching, and output sanitization — because Gemini's JSON mode does not always return clean JSON.
- A gamification layer with XP, levels, badges, and activity logs connected to a leaderboard.
- An honest account of three real problems we hit during development and how we fixed them.

D. Paper Organization

Section II covers related work. Section III describes our tech stack and why we chose it. Section IV explains the architecture. Section V goes through the implementation. Section VI covers the AI pipeline in detail. Section VII presents our evaluation. Section VIII concludes.

II. RELATED WORK

A. Online Learning Modalities and LMS Ecosystems

Platforms like Canvas, Moodle, and Blackboard have been around long enough to be genuinely mature [1]. They have plugin ecosystems, LTI integrations, gradebook workflows, and years of refinement behind them. A three-person final-year team is not going to build anything close to that and we are not trying to. What we can do is build a smaller, cleaner system that demonstrates a specific set of ideas — embedded AI assistance, role-aware content delivery, and gamified analytics within a single deployable codebase [4]. That is the trade- off and we made it deliberately.

B. Intelligent Tutoring, Feedback, and Generative Assistance

Research on intelligent tutoring has consistently shown that timely feedback and structured practice are what actually move learning outcomes [2]. The arrival of capable LLMs expanded the design space considerably, but using them in education is not straightforward. You have to constrain the output format, manage length, and keep instructors involved in what gets surfaced to students [3]. We handled this by making every generative feature return something the UI can render directly JSON for quizzes, Mermaid text for diagrams, plain text for summaries. No freeform output the frontend has to figure out on its own.

C. Learning Analytics and Gamification

Learning analytics typically means collecting activity data and turning it into dashboards that help students track progress and give instructors early signals about who is falling behind [12]. Gamification — points, levels, badges — is used to sustain engagement, though the research on whether it actually improves learning is mixed and probably context-dependent [13]. We included both. Honestly, they made the demo more compelling, and the leaderboard ended up being one of the features people interacted with most during our walkthroughs.

III. TECHNOLOGY STACK

Table ?? shows what we used. Some choices were technical decisions. Some were just practical.

Next.js made sense because the App Router lets you colo- cate API endpoints with the frontend in one repository. That simplified local development significantly — no managing two separate servers, no CORS configuration, just one thing running [4]. MongoDB with Mongoose fit our data model well. Users, activities, and subject catalogs do not need rigid relational structure, and Mongoose gave us enough schema validation without overhead.

Cloudinary was chosen for media storage because it is what most developers use for this kind of thing, the documentation is good, and the free tier covered everything we needed. We considered storing media in MongoDB as binary data early on but dropped that idea quickly — it would have made the database unwieldy and slowed down every query.

For AI we chose Google Gemini because it offers a free API. That was genuinely the deciding factor. This is a student project with no funding. OpenAI's API costs money from the first call. Gemini gave us enough quota to build and test without worrying about bills. Stability SDXL was added later as an optional image generation path it is not central to the system but it makes for a good demo.

IV. SYSTEM ARCHITECTURE

A. High-Level Overview

Fig. ?? shows the overall structure. Students interact with React pages — content catalogs, PDF viewer, dashboards, AI tools. The Next.js server handles everything that touches a secret: database operations, media uploads, AI calls. The browser only ever receives results and safe references like Cloudinary URLs.

This boundary was not something we got right immediately. Early in development we had Gemini calls going directly from the client, which meant the API key was visible in every network request. We caught it during a code review, moved everything server-side, and it has stayed that way since.

B. Domain Model

The three core Mongoose models are User, Activity, and subject entities for files and videos. The User model carries role information (student, teacher, admin), XP and level fields, badge records, and optional OAuth data for Google lo- gin. Activitystores typed learning events with timestamps quiz attempts, summaries generated, diagrams created. Subject entities store metadata and Cloudinary references for uploaded content organized by track.

We kept the schema relatively flat. Early drafts had more normalization but the dashboard and leaderboard queries got complicated fast, so we simplified.

C. Representative API Surface

Table ?? groups endpoints by responsibility. Auth, catalog, teacher uploads, analytics, AI agents, and media handling are the main groups. The actual route names in the codebase evolved a lot during development so the table communicates intent rather than exact strings.

D. Request Flow for a Generative Action

Fig. ?? shows the pattern every AI feature follows. Browser sends a request. Server authenticates. Server loads context if needed—usually extracted PDF text. Server calls Gemini.

Server validates and cleans the output. Server optionally logs an activity event. Server returns JSON. That validation step in the middle turned out to be critical, which we cover in Section VI.

E. Security and Authentication Considerations

Passwords are hashed with bcrypt before storage. JWTs are issued on login and registration. We ran into a real problem here about halfway through development. Some routes were reading the token from an HttpOnly cookie. Others expected it in the Authorization header. The result was that authenticated users were hitting 401 errors on certain analytics routes and we could not figure out why for almost two days because the requests looked fine from the browser side.

The fix was straightforward once we found it—standardize the token source per endpoint family and make sure JWT claims are consistent across login, verification, and session flows. But it was a painful two days. In a production deployment this would need a full audit before launch. Instructor-only routes also need explicit server-side role checks — hiding UI elements is not a security boundary.

V. IMPLEMENTATION

A. Application Composition

The UI has a public landing page and password-protected learning areas. Once logged in, students can access subject hubs, their dashboard, the leaderboard, and the Agents workspace where all the AI tools live. Teachers see an additional upload interface when their role allows it. We used Tailwind CSS throughout with some animation on the landing page — nothing excessive, just enough to make it look like we cared about the frontend, which we did.

B. Content Delivery and Teacher Workflow

Subject hubs bring together PDFs, downloadable files, and videos in one place organized by track. When a teacher uploads something, the system writes a metadata record to MongoDB and sends the file to Cloudinary. Those two operations need to happen in sequence — Cloudinary first, then MongoDB with the returned URL.

We learned this the hard way. The original implementation ran them in parallel to save time. What actually happened was that Cloudinary uploads occasionally took longer than expected, the MongoDB write completed first with an empty URL field, and students would open a material and get a broken link. Once we made the operations sequential — wait for Cloudinary to confirm, then write to MongoDB — the

problem went away entirely.

C. Learner Experience and Analytics

The student dashboard shows XP, current level, recent activity, and badge progress. XP gets awarded for different actions — generating a summary, completing a quiz, creating a diagram. The exact weights are configurable. The leaderboard ranks students by total XP and updates in real time as activity events come in. We added the leaderboard mostly to make the demo more engaging. It ended up being the feature that got the most reaction during walkthroughs — people immediately wanted to know where they ranked, which told us the gamification actually worked on some level.

D. Client State and Initialization

Auth state is managed client-side using a store that initializes from persisted session data. Protected routes check this state before rendering. The key thing we learned here is that you need three states, not two. Not just authenticated and unauthenticated — you also need a loading state for the moment between page load and when the store has finished checking the session. Without it the UI briefly flashes the unauthenticated view on every refresh, which looks broken even when the user is fully logged in.

Server routes always validate credentials independently. The client-side check is only for UX — it is not where security lives.

E. Error Handling and Observability

Route handlers return structured error messages for the failures we ran into most: missing environment variables, up- stream AI errors, unparseable model output, and auth failures. Logging right now is mostly consoling output, which is fine for a demo environment. Before deploying this anywhere real you would want centralized logging and request tracing so you can actually debug failures in production.

VI. AI INTEGRATION PIPELINE

A. LLM Service Layer

The AI layer is a server-side Gemini helper that maintains an ordered list of model candidates, uses JSON response mode when the task needs structured output, and caches results in memory with a TTL and a maximum entry count. We added the cache after watching students run the same summarization request on the same PDF multiple times in a single session. Every duplicate request was burning API quota for identical output. The cache fixed that and also made the tool feel faster on repeated use.

B. PDF-Aware Learning Assistance

When a student requests help with a reading, the server fetches the PDF, extracts text from it, and passes that text to Gemini. We kept extraction as a completely separate step from the LLM call. This made the pipeline easier to debug — if something went wrong, we could tell immediately whether the problem was in extraction or in generation. It also meant we could reuse the same extracted text across multiple agent tasks in one session without hitting the PDF again.

Fig. ?? shows the four steps: fetch the PDF reference, extract text on the server, pass to Gemini with the appropriate task prompt, return structured output to the UI.

C. Agent Modules

Quiz agent was the one that gave us the most trouble. We prompt Gemini to return a JSON object with a title and a list of questions. In theory the JSON response mode handles this. In practice Gemini sometimes returned perfectly valid JSON wrapped inside a markdown code block like this:

‘‘‘json
{ "title": "...", "questions": [...] } ‘‘‘

Our JSON parser threw an error every time that happened because the backticks are not valid JSON. We added a sanitization step that strip markdown fences before parsing, and that fixed it. It is a small thing but it took longer to notice than it should have because the error message was not obvious about what was wrong.

Chart agent returns Mermaid diagram syntax. The frontend renders it directly. No parsing issues here — plain text in, diagram out.

Email agent takes a short description of what the student wants to say and returns a drafted professional message. Simple and it works well.

Speech flows pass multimodal inputs depending on the configuration. Optional SDXL handles image generation fol- lowing Stability parameters [7] — it is

slow but produces good results for the demo.
Table ?? maps each task to its output type.

D. Algorithmic Sketch (Server-Side Generation)
Algorithm 1 is the cache-first pattern the service layer uses for every generation request.
Algorithm 1 Cache-first generation with model fallbacks
1) Input: task label *t*, prompt text *p*, JSON mode flag *j*.
2) Compute cache key *k* = hash(*t*, prefix(*p*)).
3) If a valid entry exists for *k* in cache, return it immediately.
4) Otherwise, for each model *m* in the candidate list:
a) Call GeminiGenerate(*m, p, j*) to get output *y*.
b) If *y* is non-empty, store in cache with expiry and return *y*.
5) If every candidate fails, return a task-specific fallback string.

VII. RESULTS AND EVALUATION

A. Evaluation Goals and Limitations
We evaluated three things: whether the main features work end-to-end, what the latency looks like across different route types, and what people actually said when they used it. We are not measuring learning outcomes. That requires a controlled study with proper instruments and ethics approval — neither of which was feasible for a final-year project in this timeline.

B. Experimental Setup
Latency numbers were collected on a development ma- chine. Around 20 manual runs per route category with the cache cleared between runs. Network varied between home broadband and campus Wi-Fi. The numbers capture server- side round trips including provider latency. Human interaction time is not included.

C. Functional Validation
We tested every feature end-to-end with two of us acting as testers — one playing student, one playing instructor. A feature passes when the output matches what the UI expects and no error appears. Table ?? lists each scenario in the order we would run them in a live demo, starting with authentication and ending with image generation.
Three issues came up during validation that we fixed before finalizing. First, some routes were missing environment vari- able checks and failed silently instead of returning a useful error — easy fix, just took a pass through every handler. Second, JWT claims were inconsistent between the login route and the analytics routes, which caused the dashboard to return 401 errors for fully authenticated users. That was the two- day debugging session mentioned earlier. Third, Gemini's quiz output occasionally came back as JSON inside a markdown block, breaking the renderer — fixed with the sanitization step described in Section VI.

D. Latency Observations
Fig. ?? shows the mean timings. Database-only routes like the dashboard API and materials list stay under half a second. Generative routes — summarization, quiz generation, chart generation — land between two and three seconds, which is dominated by Gemini response time. SDXL image generation averages around 13 seconds because of the diffusion sampling process plus the Cloudinary upload at the end. That is slow but users did not react badly to it when the UI showed a progress indicator.

E. Qualitative Observations
Three things came up consistently during walkthroughs. People liked having AI assistance next to the reading — the reaction was noticeably different from using a separate chatbot tab. Quiz outputs sometimes needed a second look — Gemini occasionally generated questions that were too easy or slightly misaligned with the uploaded material, which is something an instructor review step would catch. And latency tolerance was much higher when the UI communicated that something was happening. A three-second wait with a spinner felt fine. The same wait with a frozen screen felt broken.

F. Comparison to Baseline LMS Features
Table ?? shows where this system sits relative to typical LMS expectations. We are not trying to compete with Canvas or Moodle. The table is just context for reviewers who want to know what is and is not here.

VIII. CONCLUSION

We built something that works. That sounds like a low bar but for a system that touches authentication,

cloud media, database persistence, and live AI generation across multiple agent types — getting it all working together took real effort. The things that surprised us most were the small integration problems. Gemini not returning clean JSON. Cloudinary uploads racing with database writes. JWT claims that looked fine on one route and broke another. None of these are conceptually difficult but they all took time to find and fix, and none of them showed up in any tutorial or documentation we read.

If someone wanted to take this further the most important next steps would be adding an instructor review layer before AI-generated quizzes go live, streaming Gemini output instead of waiting for the full response, and doing a proper audit of the authentication flow before any real deployment. The gamification layer could also be more sophisticated — right now XP weights are fixed and there is no opt-out for the leaderboard, which raises fairness questions worth thinking through.

We learned more building this than we expected to. That is probably the most honest thing we can say about it.

IX. FUTURE WORK

1) Standardize authentication so every route uses the same token source and claim structure.
2) Add instructor review step for AI-generated assessments before student access.
3) Stream Gemini output to reduce perceived latency on summarization and quiz routes.
4) Run a proper user study with pre and post measurement instruments.
5) Privacy review for the PDF processing pipeline — right now extracted text is not persisted but that should be formally verified.
6) Rate limiting and cost monitoring before any public deployment.

REFERENCES

[1] S. Hrastinski, "Asynchronous and synchronous e-learning," *Educ. Quarterly*, vol. 31, no. 1, pp. 51–55, 2008.

[2] J. Wei *et al.*, "Emergent abilities of large language models," *Trans. Mach. Learn. Res.*, 2022.

[3] T. Brown *et al.*, "Language models are few-shot learners," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 33, 2020, pp. 1877–1901.

[4] Vercel, "Next.js documentation," 2025. [Online]. Available: https://nextjs.org/docs

[5] MongoDB Inc., "Mongoose documentation," 2025. [Online]. Available: https://mongoosejs.com/docs/guide.html

[6] Google, "Gemini API documentation," 2025. [Online]. Available: https://ai.google.dev/gemini-api/docs

[7] Stability AI, "Platform API reference," 2025. [Online]. Available: https://platform.stability.ai/docs/api-reference

[8] Cloudinary, "Cloudinary documentation," 2025. [Online]. Available: https://cloudinary.com/documentation

[9] M. West, "Education technology can help address inequality in a high inequality world," Brookings Institution, TechTank blog, 2022.

[10] M. Jones, J. Bradley, and N. Sakimura, "JSON web token (JWT)," RFC 7519, IETF, May 2015. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7519

[11] N. Provos and D. Mazieres, "A future-adaptable password scheme," in *Proc. USENIX Annu. Tech. Conf. (ATEC)*, 1999, pp. 81–91.

[12] Siemens and P. Long, "Penetrating the fog: Analytics in learning and education," *EDUCAUSE Rev.*, vol. 46, no. 5, pp. 30–32, 2011.

[13] S. Dicheva *et al.*, "Gamification in education: A systematic mapping study," *Educational Technology & Society*, vol. 18, no. 3, pp. 75–88, 2015.

[14] NIST, "Digital identity guidelines: Authentication and lifecycle management," NIST Special Publication 800-63B, Jun. 2017. [Online]. Available: https://pages.nist.gov/800-63-3/sp800-63b.html

[15] Meta Open Source, "React documentation," 2025. [Online]. Available: https://react.dev